

# Delegaty i zdarzenia w C#

# O czym będzie wykład?

1. Co to takiego delegat ?
2. Co jeszcze może delegat ?
3. Synchroniczne i asynchroniczne realizowanie metod delegata
4. Metody anonimowe.
5. Delegaty a wyrażenia lambda.
6. Delegaty i wyrażenia lambda w LINQ.
7. Delegaty i zdarzenia.
8. Podsumowanie.

## Co to takiego delegat ?

- Mając w pamięci poprzedni wykład dotyczący m. in. generyczności narzuca się pewien wniosek, iż twórcy języków programowania starają się udostępniać programistom różne rozwiązania ułatwiające i przyspieszające tworzenie finalnego kodu, np. przez możliwość parametryzacji typów, tzw. generyczność.
- Innym rozwiązaniem dającym programiście możliwość tworzenia elastycznego kodu i jego optymalizację są tzw. **delegaty**
- Historycznie podchodząc do istoty delegatów w C# należy cofnąć się do **języka C**, w którym istotnym elementem są wskaźniki, w tym **wskaźniki na funkcje**.
- Wskaźniki są wygodnym, ale niebezpiecznym narzędziem. Stosując je nie możemy liczyć na jakąkolwiek kontrolę tego co robimy. Za ich pomocą możemy „zwiedzić” cały obszar przydzielonej nam przez system operacyjny pamięci. Nie zawsze kończy się taka „wycieczka” pomyślnie.
- Biorąc pod uwagę te mankamenty wskaźników i wskaźników do funkcji twórcy języka C# postanowili ograniczyć potrzebę korzystania z wskaźników lub ich stosowanie poddać pewnemu nadzorowi. Pomysłem na nadzorowane i bezpieczne stosowanie wskaźników do funkcji stały się **delegaty**.
- Jednak delegaty są bezpieczniejsze, dana delegata może tylko referować się do metody, której sygnatura jest zgodna z delegatą. Nie możesz wywołać delegaty, która nie referuje się do właściwej dla siebie metody.

# Co to takiego delegat ?

- **Delegaty** w C# są klasami dziedziczącymi po klasie bazowej **System.Delegate** . Ich definiowanie odbywa się wg poniższego wzorca:

*Modyfikator dostępu **delegate** Typ\_Wynikowy Identyfikator ([parametry]);*

Gdzie: *Modyfikator dostępu - public, private, itd.*

*Identyfikator - nazwa nadana delegatowi*

*parametry - lista parametrów metody, która może reprezentować delegat*

*Typ\_Wynikowy - typ wartości zwracanej przez metodę reprezentowaną przez delegata;*

Przykład deklaracji i zastosowania delegata:

*Using System;*

*Public class Program*

*{*

*public delegate void Delegacja(); // delegat może reprezentować każdą  
// metodę bezparametrową, nie zwracającą  
// żadnej wartości*

## Co to takiego delegat ?

```

public static void Metoda1() // metoda bezparametrowa
{
    Console.WriteLine(„Została wywołana metoda Metoda1.”);
}
public static void Metoda2(string napis) // ta metoda posiada parametr
{
    Console.WriteLine(napis);
}
public static void Main()
{
    Delegacja del1 = new Delegacja(Metoda1);
    //Delegacja del2 = new Delegacja(Metoda2);
    del1(); // teraz obiekt del1 reprezentuje metodę Metoda1
    //del2(„Została wywołana metoda Metoda2.”);
}
}

```

Jak widać w przykładzie delegaty są bezpieczniejsze od wskaźników na funkcję w języku C/C++, gdyż obiekt utworzony w oparciu o definicję delegata może „reprezentować” tylko metody o liście parametrów oraz typie zwracanej wartości zgodne z definicją delegata.

# Co to takiego delegat ?

W praktyce zapisem wygodniejszym od

```
Delegacja del1 = new Delegacja(Metoda1);
```

i akceptowanym przez kompilator jest:

```
Delegacja del1 = Metoda1;
```

W tym momencie można zadać pytanie:

*Po co tworzyć jakiś nowy byt, który robi to samo co metoda, którą reprezentuje ???*

Odpowiedź brzmi:

*Ale ten byt (obiekt) może teraz reprezentować każdą metodę spełniającą warunki deklaracji delegata.*

W kodzie naszej klasy może dodać np. metodę *Metoda3*:

```
public static void Metoda3()
{
    Console.WriteLine(„Została wywołana metoda Metoda3”);
}
```

# Co to takiego delegat ?

Kod w metodzie *Main()* może mieć teraz postać:

```
public static void Main()
{
    Delegacja del1 = Metoda1;
    del1();           // obiekt del1 „wykonuje” metodę Metoda1;
    del1 = Metoda3;   // obiekt del1 „reprezentuje” metodę Metoda3
    del1();           // obiekt del1 „wykonuje” metodę Metoda3;
}
```

Podsumowując to wprowadzenie można stwierdzić, że **delegaty** to:

- bardzo podobna konstrukcja do przypisywania wartości polom, tyle że wartością przypisywaną jest **metoda**.
- Kolejny po **polach**, **metodach**, **właściwościach**, **konstruktorach** itd. pełnoprawny element, za pomocą którego budować możemy klasy.



# Co jeszcze może delegat?

## 1. Delegat jako funkcja zwrotna:

Czasami zachodzi potrzeba użycia w klasie **kodu zewnętrznego** (chcemy za pomocą parametru przekazać kod jakiejś metody). Przydatność delegatów w takiej sytuacji zaprezentujemy na prostym przykładzie:

Załóżmy, że utworzyliśmy klasę o nazwie **KlasaR**, która posiada dwie właściwości: **W1** i **W2**. Pierwsza będzie typu **int**, a druga typu **double**. Klasa będzie też zawierała metodę **WykonajR** której zadaniem będzie wykonanie metody(metod) przekazanych przez parametr typu delegata **DelegatWyswietl**:

namespace Przyklady

```
{
    public delegate void DelegatWyswietl(KlasaR obj); // delegat dla metod o parametrze klasy KlasaR
    public class KlasaR
    {
        public int W1
        {
            get { return 100; }
        }
        public double W2
        {
            get { return 2.14; }
        }
        public void WykonajR(DelegatWyswietl del) // ta metoda wykona metodę podaną parametrem
        {
            del(this);
        }
    }
}
```



# Co jeszcze może delegat?

## 1. Delegat jako funkcja zwrotna (cd.):

```
class Program
{
    public static void Wyszwietl(KlasaR obj) // zdefiniowaliśmy sobie taką metodę w klasie program
    {
        Console.WriteLine(obj.W1);
        Console.WriteLine(obj.W2);
    }

    static void Main(string[] args)
    {
        KlasaR d = new KlasaR();           // tworzymy obiekt klasy KlasaR (zdefiniowanej wcześniej)
        DelegatWyszwietl del= Wyszwietl;   // tworzymy obiekt delegata DelegatWyszwietl i przypisujemy
                                           // zdefiniowaną powyżej metodę Wyszwietl
        d.WykonajR(del);                    // uruchamiamy metodę WykonajR obiektu d (klasy
                                           // KlasaR, poprzez parametr del „wstrzykujemy” kod zewnę-
                                           // trznej w stosunku do klasy KlasaR metody Wyszwietl
    }
}
```

# Co jeszcze może delegat?

## 2. Multicast – wskazywanie na kilka funkcji:

- Delegacja pozwala na powiązanie zdarzenia nawet z kilkoma metodami.
- W związku z powyższym musi istnieć możliwość przypisania jej większej liczby metod niż tylko jednej, tak jak miało to miejsce w dotychczasowych przykładach.
- Tak jest w istocie - **każdy obiekt delegacji może być powiązany z dowolną liczbą metod**, można je do tego obiektu zarówno dodawać jak i odejmować (odejmowanie oznacza w tym przypadku usunięcie powiązania delegacji z daną metodą).

Oczywiście należy pamiętać, że **każda powiązana metoda musi mieć deklarację zgodną z deklaracją delegacji**. Prosty przykład zostanie zaprezentowany na kolejnych slajdach.

# Co jeszcze może delegat? (multicast)

```
class Program
{
    public static void WyświetlW1(KlasaR obj)
    {
        Console.WriteLine("Wartosc W1 to {0}.", obj.W1);
    }
    public static void WyświetlW2(KlasaR obj)
    {
        Console.WriteLine("Wartosc W2 to {0}.", obj.W2);
    }
}
```

- ✖ W klasie **Program** znalazły się dwie metody wyświetlające dane z obiektów typu **KlasaR** (patrz przykład dot. funkcji zwrotnych): **WyświetlW1** i **WyświetlW2**. Pierwsza wyświetla wartość właściwości **W1**, a druga właściwości **W2**. W metodzie **Main** (kod na kolejnym slajdzie) powstał nowy obiekt typu **KlasaR** oraz dwie delegacje typu **DelegatWyświetl** : **del1** i **del2**. Pierwszej została przypisana metoda **WyświetlW1**, a drugiej – **WyświetlW2**.

# Co jeszcze może delegat? (multicast)

```
static void Main(string[] args)
{
    KlasaR d = new KlasaR();
    DelegatWyswietl del1 = WyswietlW1;
    DelegatWyswietl del2 = WyswietlW2;

    DelegatWyswietl del3 = del1 + del2;
    d.WykonajR(del3);
    Console.WriteLine(" - ");
    del3 -= del2;
    d.WykonajR(del3);
    Console.WriteLine("- ");
    del3 += del2;
    d.WykonajR(del3);
    Console.WriteLine("- ");
    del3 += del2;
    d.WykonajR(del3);
}
}
```

Ciekawa jest instrukcja wyświetlająca trzecią delegację **del3**. Wygląda to jak dodawanie delegacji, a oznacza utworzenie obiektu **del3** i przypisanie mu wszystkich metod powiązanych z delegacjami **del1** i **del2**. Skoro więc obiekt **del1** był powiązany z metodą **WyswietlW1**, a **del2** z metodą **WyswietlW2**, to **del3** będzie powiązany zarówno z metodą **WyswietlW1**, jak i **WyswietlW2**. Przekonujemy się o tym dzięki wywołaniu.

- ✖ Faktycznie spowoduje ono uruchomienie obu metod.
- ✖ Skoro delegacje można dodawać, to można je też odejmować. Oczywiście oznacza to usunięcie z delegacji **del3** wszystkich odwołań do metod występujących w delegacji **del2**. Dlatego też funkcja **del3** spowoduje wywołanie jedynie metodą **WyswietlW1**.
- ✖ Nic nie stoi na przeszkodzie, aby jedna metoda została dodana do delegacji kilka razy. Spowoduje ona, że w deklaracji **del3** znajdą się trzy odwołania.

# Co jeszcze może delegat? (multicast)

```

C:\ file:///G:/szkola/Języki i Paradygmaty Programowania 4/2012/Wyklady/Wyklad 2/Materialy...
Wartosc w1 to 100.
Wartosc w2 to 2,14.
-----
Obiekt del3 zawiera metody WyświetlW1 oraz WyświetlW2
-----
Wartosc w1 to 100.
-----
Z obiektu del3 usunięto metodę WyświetlW2
-----
Wartosc w1 to 100.
Wartosc w2 to 2,14.
-----
Do obiektu del3 ponownie dodano metodę WyświetlW2
-----
Wartosc w1 to 100.
Wartosc w2 to 2,14.
Wartosc w2 to 2,14.
-----
Do obiektu del3 dodano jeszcze raz metodę WyświetlW2
-----

```

Powyżej obraz praktycznego zastosowania **funkcji zwrotnych** i **multikastów** z naszego przykładu (do obiektu **d** klasy **KlasaR** „wstrzykiwane” są określone zestawy metod zewnętrznych **WyświetlW1** i **WyświetlW2** za pomocą obiektu **del3** delegata **DelegatWyświetl**).

## Co jeszcze może delegat? (multicast)

Metody do delegacji mogą być też dodawane lub odejmowane bezpośrednio, a nie jak w przedstawionym przykładzie – za pomocą innych delegacji. Oznacza to, że można utworzyć np. delegację **del3** i przypisać jej metodę **WyświetlW2**:

```
DelegatWyświetl del3 = WyświetlW2;
```

a następnie dodać do niej metodę **WyświetlW2**:

```
del3 += WyświetlW2;
```

i w dalszej części kodu, gdy znajdzie taka potrzeba usunąć z delegacji del metodę **WyświetlW2**:

```
del3 -= WyświetlW2;
```

Widać więc, że C# jest tutaj bardzo przyjazny dla programisty.



# Delegaty – praca synchroniczna a asynchroniczna

W prezentowanych wcześniej przykładach dodawaliśmy (rejestrOWaliśmy) do obiektu typu **delegat** kolejne metody o sygnaturze zgodnej z wzorcową sygnaturą delegata **DelegatWyswietl** :

```
DelegatWyswietl del3 = del1 + del2;  
del3 += WyswietlW4;  
del3 += WyswietlW5;  
d.WykonajR(del3);
```

Wykonanie ich nastąpi **synchronicznie**, czyli w takiej kolejności jak zostały zarejestrowane., a więc metoda WyswietlW5 będzie wykonywana dopiero po zakończeniu realizacji metody WyswietlW4. Takie podejście rodzi dwa problemy:

- metoda może się „zawiesić” i nigdy nie zwrócić sterowania,
- może się wykonywać bardzo długo, blokując wykonanie kolejnych metod.

Stosując zaprezentowany tutaj mechanizm rejestrowania w obiekcie delegata listy metod nie mamy możliwości reagowania na wymienione powyżej niedogodności. W celu rozwiązania tego problemu na platformie .NET stworzono mechanizm umożliwiający **asynchroniczne** uruchamianie metod zarejestrowanych w obiekcie typu delegata. To zagadnienie zostanie omówione w trakcie wykładów dot. wielowątkowości.



# Metody anonimowe

Z pojęciem delegata związane jest pojęcie **metody anonimowej** (*Anonymous methods*). Jest to metoda, której **dziwactwo** polega na tym, że **nie posiada swojej nazwy**. Zachodzi więc pytanie – jak możliwa jest jej identyfikacja? Otóż metoda taka jest w momencie definiowania kodu wiązana z pewnym delegatem. Jest to wygodne rozwiązanie w przypadku potrzeby zdefiniowania metod o krótkim i prostym kodzie, które są tak proste, że wręcz nie „zasługują” na jakąś nazwę. Metodę tą będziemy mogli wywołać tylko korzystając z delegacji.

Proszę sobie przypomnieć jak przy okazji omawiania multikastu zdefiniowaliśmy pewien obiekt **del3** i przypisaliśmy mu pewną metodę **WyświetlW2**:

*DelegatWyświetl del3 = WyświetlW2;*

W tym momencie obiekt **del3** reprezentuje metodę **Wyświetl2**. Czy więc nie można postąpić jak w przedstawionym poniżej kodzie?

```
DelegatWyświetl del5 = delegate(KlasaR obj) // definiujemy obiekt del5
// i przypisujemy mu kod
{
    Console.Write(" Wartość w1 to {0}.", obj.W1);
    Console.WriteLine(" - wykonane przez metodę anonimową");
};
del3 += del5;
d.WykonajR(del3);
```

## Metody anonimowe (cd.)

W kodzie zaprezentowanym na poprzednim slajdzie niejako „z marszu”, znając sygnaturę metod jakie mogą wykonywać obiekty delegatów typu *DelegatWyświetl*, można utworzyć obiekt takiego delegata i przypisać mu pewien kod. Mamy więc kolejne udogodnienie jakie wnoszą delegaty do pisania elastycznego i „wariantowalnego” kodu.

Poniżej listing wcześniej prezentowanej aplikacji wzbogacony o prezentowany przykład zastosowania metody anonimowej:

```

C:\ file:///G:/szkola/Języki i Paradygmaty Programowania 4/2012/Wyklady/Wy
Wartosc w1 to 100.
Wartosc w2 to 2,14.

-----
Obiekt del3 zawiera metody WyświetlW1 oraz WyświetlW2

-----
Wartosc w1 to 100.

-----
Z obiektu del3 usunięto metodę WyświetlW2

-----
Wartosc w1 to 100.
Wartosc w2 to 2,14.

-----
Do obiektu del3 ponownie dodano metodę WyświetlW2

-----
Wartosc w1 to 100.
Wartosc w2 to 2,14.
Wartosc w2 to 2,14.

-----
Do obiektu del3 dodano jeszcze raz metodę WyświetlW2

-----
Wartosc w1 to 100.
Wartosc w2 to 2,14.
Wartosc w2 to 2,14.
Wartosc w1 to 100. - wykonane przez metodę anonimową

-----
Do obiektu del3 dodano metodę anonimową
  
```

# Delegaty a wyrażenia lambda

- W dotychczasowych rozważaniach poznaliśmy pewne specyficzne rozwiązania w języku C#, które mają na celu umożliwienie programiście tworzenie kodu w sposób elastyczny, poprzez różnego rodzaju rozwiązania umożliwiające ogólnie rzecz biorąc jego parametryzację, czy to typów (**generyczność**), czy używanych metod (**delegaty**). Ciekawym rozwiązaniem, wprowadzonym w wersji 3.0 języka C# są **wyrażenia lambda**, które jako efekt ich działania nie zwracają, tak jak metody wartość, ale właśnie metodę.
- Takie rozwiązanie – zwracanie przez metodę kodu jakiejś metody, jest zaczerpnięte z paradygmatu **programowania funkcyjnego** wykorzystującego **rachunek lambda**. Szczegóły paradygmatu programowania funkcyjnego omówimy w następnym semestrze. Istotne jest w tym momencie to, że użyteczne języki programowania nigdy nie są realizacją dokładnie jednego paradygmatu programowania, lecz wykorzystują szereg elementów innych paradygmatów.
- Jak pamiętamy typowa metoda musi składać się z 4 elementów:
  - Nazwy
  - Listy parametrów
  - Ciała metody
  - Zwracanej wartości

W przypadku metod anonimowych zrezygnowano z nazwy, natomiast w wyrażeniach lambda dodatkowo ze zwracanej wartości.

# Delegaty a wyrażenia lambda

- Symbolem wyrażenia lambda jest złożenie znaków = oraz >, czyli =>
- W istocie, kiedy przyjrzymy się tej konstrukcji zauważymy, że jest ono bardzo podobne do metod anonimowych – umożliwia tworzenie jeszcze prostszego kodu.
- Kontynuując poprzedni przykład zdefiniujmy sobie **wyrażenie lambda**:

```
DelegatWyswietl del6 = (KlasaR obj) => // definiujemy obiekt del6
                                     // i przypisujemy mu kod
{
    Console.Write(" Wartość w2 to {0}.", obj.W2);
    Console.WriteLine(" - wykonane przez wyrażenie lambda");
};
del3 += del6;
d.WykonajR(del3);
```

Ponieważ w wyrażeniach lambda kładzie się nacisk na uproszczenie kodu, więc twórcy języka C# dopuszczają też możliwość poniższego zapisu **wyrażenia lambda**:

```
DelegatWyswietl del6 = obj => {
    Console.Write(" Wartość w2 to {0}.", obj.W2);
    Console.WriteLine(" - wykonane przez wyrażenie lambda");
};
```

## Delegaty a wyrażenia lambda

Przykład na poprzednim slajdzie był rozwinięciem wcześniejszych przykładów dot. delegatów i być może nie pokazywał istoty pomysłu na wyrażenia lambda, a więc maksymalnego skrócenia i uproszczenia kodu. Poniższy przykład pokaże to dobitniej:

```
delegate int operacjeMatematyczneDel(int x);    // delegat metod jednoargumentowych
delegate int operacjeMatDwuArgDel(int x, int y); // dwuargumentowych

static void Main(string[] args)
{
    operacjeMatematyczneDel operacjeMatematyczne = null; // tworzymy obiekt delegata
    operacjeMatematyczne += (x => x * x); // to wyrażenie zwraca kwadrat wartości argum. X
    operacjeMatematyczne += (x => {return x * x}); // można to zapisać tak
    operacjeMatematyczne += ((int x) => {return x * x}); // można też tak

    operacjeMatDwuArgDel operacjeMatDwuArg = null;
    operacjeMatDwuArg += ((x, y) => { x++; y--; return x / y; });
}
```



# Delegaty a wyrażenia lambda

## Czas na podsumowanie:

- Wyrażenia lambda pojawiły się w wersji 3 .0 C# i są naturalnym następcą metod anonimowych. Ich pojawienie stało się jednym z ważnych elementów we wprowadzonej w wersji .NET 3.5 technologii **LINQ** (*Language INtegrated Query*). W stosunku do metod anonimowych umożliwiają tworzenie prostszego, bardziej zwartego zapisu poleceń.
- Ciało wyrażenia lambda może być prostym wyrażeniem albo blokiem kodu C# z wieloma poleceniami, wywołaniami innych metod, definicjami pól itd. (w metodach anonimowych może być tylko blokiem kodu).
- Wyrażenia lambda mogą zwracać wartości jednak muszą one pasować do typu delegata do którego są dodawane.
- Pola zdefiniowane wewnątrz wyrażenia lambda istnieją tylko w tym bloku kodu i znikają gdy metoda się skończy.
- Wyrażenie lambda ma dostęp do wszystkich pól i metod znajdujących się po za tym wyrażeniem.
- Wyrażenia lambda pozwalają, aby typy parametrów były pomijane i wnioskowane w miejscach gdzie metody anonimowe wymagają jawnie podanych typów parametrów .
- Wyrażenie lambda może zmienić wartości pól zewnętrznych jeśli są one przesłane do ciała wyrażenia lambda za pomocą słów kluczowych **ref** lub **out**.

# Delegaty i wyrażenia lambda w LINQ

**LINQ** (*Language INtegrated Query*) - to część technologii .NET, opracowana przez **Andreas Hejlsberg** - znanego z zaprojektowania języków **Delphi** i **C#**, wprowadzona w wersji 3.5 .NET. Technologia **LINQ** umożliwia zadawanie pytań na obiektach. Składnia języka LINQ jest prosta i przypomina SQL. Wykorzystywane są w nim zarówno **delegaty** jak i **wyrażenia lambda**.

## Czym w istocie jest LINQ :

**LINQ** stanowi warstwę abstrakcji nad różnymi źródłami danych. Baza danych i jej elementy **traktowane są jak obiekty**. Przestrzeń które obsługuje LINQ to:

- Obiekty implementujących interfejs **IEnumerable<T>**
- Bazy danych
- Język **XML**

**LINQ** posiada pełne wsparcie dla transakcji, widoków, procedur składowanych, itd. Zapytania stają się częścią języków na platformie .NET (**C#**, **VisualBasic**, **C++**, **Delphi** itd...).



# Delegaty i wyrażenia lambda w LINQ

Przykładowymi standardowymi operatorami LINQ są np.:

- **Select** - operator ten jest używany by wybrać z kolekcji odpowiedniego rodzaju dane (zbiory/podzbiory danego obiektu) .
- **SelectMany** - jest używany by dokonać wyświetlenia w przypadku relacji jeden-do-wielu - czyli jeśli obiekt w kolekcji zawiera inną kolekcję jako część danych , **SelectMany** będzie użyte do wybrania całej pod-kolekcji.
- **Where** - pozwala zdefiniować zbiór zasad dla każdego obiektu w kolekcji i wszystkie te obiekty, które nie pasują do wybranej reguły są odfiltrowywane.
- **Sum** - używany do otrzymywania sumy .
- **Min** - używany do otrzymania minimalnej wartości.
- **Join** - operator **Join** to operator bazujący na dwóch kolekcjach i tworzy z nich obiekt wynikowy.
- **OrderBy** - sortuje wyniki po wybranym elemencie kolekcji - według określonego klucza.

Każde z tych poleceń jest w trakcie kompilacji **zamieniane** najpierw na odpowiadające im **wyrażenia lambda**.

# Delegaty i wyrażenia lambda w LINQ

## Przykład zapytania LINQ:

W przykładzie chodzi o wybranie w zapytaniu wszystkich obiektów z właściwością *SomeProperty* mniejszą niż 10.

```
int someValue = 5;
var results = from c in someCollection
               let x = someValue * 2
               where c.SomeProperty < x
               select new {c.SomeProperty, c.OtherProperty};
foreach (var result in results)
{
    Console.WriteLine(result);
}
```

Ostatecznie kompilator wygeneruje klasę:

```
IEnumerable<SomeOtherClass> results = SomeCollection.Where
(
    c => c.SomeProperty < (SomeValue * 2)
)
.Select
(
    c => new {c.SomeProperty, c.OtherProperty}
);
foreach (SomeOtherClass result in results)
{ Console.WriteLine(result.ToString()); }
```

# Delegaty i zdarzenia

## Mały wstęp:

- Delegaty umożliwiają reprezentowanie wielu metod o takiej samej sygnaturze przez jednego przedstawiciela (tzn. delegata), ale wciąż musi być on wywołany jawnie. Z programistycznego punktu widzenia dobrze byłoby, aby **delegaty uruchomiały się automatycznie** gdy coś ważnego się wydarzy – ma miejsce np. **zdarzenie**.
- **Zdarzenia** w C# są elementem języka **pozwalającym na komunikacje pomiędzy obiektami** (klasami), będącymi ze sobą w jakiś sposób zależne (np. obiekty obsługujące mysz oraz formularz w aplikacji okienkowej często współpracują ze sobą).
- Obiekt może zgłosić **zdarzenie** (np. obiekt obsługujący myszkę zgłosi kliknięcie w któryś z jej klawiszy lub przemieszczenie w osi x lub y), które zostanie przekazane do innego obiektu (np. obiektu formularza), który nazywany jest **subskrybentem (subscriber)**. Obiekt generujący zdarzenie nazywamy **publikatorem (publisher)**.
- Zadaniem **publikatora** jest określenie zgłaszanego zdarzenia, natomiast **subskrybent** jest odpowiedzialny za podjęcie odpowiedniej akcji.
- Każde zdarzenie może posiadać wielu subskrybentów (np. kliknięcie myszki), zaś subskrybent może obsługiwać zdarzenia z różnych publikatorów (np. formularz zdarzenia z myszki lub klawiatury).

# Delegaty i zdarzenia

Aby skorzystać z obsługi zdarzeń należy to zdarzenie zdefiniować za pomocą słowa kluczowego **event** w klasie **publikatora**. Jednak wcześniej należy utworzyć odpowiadającą zdarzeniu delegację, a zatem postępowanie jest dwuetapowe:

1. Definiujemy delegata:

*[modyfikator\_dostępu] typ\_zwracany delegate nazwa\_delegacji(argumenty);*

np.: `public delegate void MojDelegat(int liczba);`

2. Definiujemy jedno lub więcej zdarzeń w postaci:

*[modyfikator\_dostępu] event nazwa\_delegacji nazwa\_zdarzenia;*

np.: `public event MojDelegat MojeZdarzenie;`

W naszym przykładzie (pełny kod klasy publikującej na następnym slajdzie) zgłoszenie zdarzenia **MojeZdarzenie** przez publikator jest równoznaczne z wykonaniem pewnej metody (zestawu metod) związanych z obiektem typu delegata **MojDelegat**.

# Delegaty i zdarzenia

## Kod klasy publikującej:

```
public class KlasaPublikujaca
{
    //definicja delegata
    public delegate void MojDelegat(int liczba);

    //definicja zdarzenia, które jest obsługiwane przez delegata MojDelegat
    public event MojDelegat MojeZdarzenie;

    //pewna metoda dodająca dwie liczby całkowite, generująca w określonym
    // wypadku zdarzenie
    public int Dodaj(int a, int b)
    {
        //jeśli warunek spełniony, to zachodzi zdarzenie MojeZdarzenie
        if (a + b > 50)
        {
            MojeZdarzenie(a + b); // to zdarzenie spowoduje uruchomienie metody
                                   //reprezentowanej przez naszego delegata
        }

        return a + b;
    }
}
```

# Delegaty i zdarzenia

**Kod prezentujący wykorzystanie zgłoszenia zdarzenia przez obiekt subskrybenta:**

```
class GlownaKlasa
{
    // Metoda możliwa do „reprezentowania” przez delegat MojDelegat
    static void PewnaMetoda(int liczba)
    {
        Console.WriteLine("Wynik otrzymaliśmy dzięki delegatowi. Suma wynosi: {0}.",
            liczba.ToString());
    }

    public static void Main()
    {
        // tworzymy obiekt publikujący
        KlasaPublikujaca kp = new KlasaPublikujaca();

        // do zdarzenia MojeZdarzenie obiektu kp „podpinamy” metodę PewnaMetoda subskrybenta
        kp.MojeZdarzenie += new KlasaPublikujaca.MojDelegat(PewnaMetoda);

        // w tym przypadku użycie metody kp.Dodaj spowoduje wykonanie metody PewnaMetoda
        Console.WriteLine("Wynik sumy to: {0}", kp.Dodaj(44, 88));
    }
}
```

## Delegaty i zdarzenia

W przedstawionym na poprzednich slajdach przykładzie wykorzystania zdarzeń w aplikacji konsolowej można by po stosunkowo prostych modyfikacjach uzyskać **efekt obsługi kliknięcia w któryś klawisz myszki** i wyświetlenia w okienku komunikatu za pomocą kontrolki typu **label**.

W tym przypadku metodę **Dodaj(int a, int b)** w klasie **KlasaPublikująca** zastąpiono by metodą **KliknięcieKlawiszaMyszy(bool sygnal)**. Oczywiście zmienić trzeba było by też sygnaturę delegata. Kod tej metody mógłby wyglądać tak:

```
public void KlikniecieKlawiszaMyszy(bool sygnal)
{
    //jeśli warunek spełniony, to zachodzi zdarzenie MojeZdarzenie
    if (sygnal == true)
    {
        MojeZdarzenie(sygnal); // to zdarzenie spowoduje uruchomienie metody
                               //reprezentowanej przez naszego delegata
    }
}
```

Metoda powyższa byłaby na formularzu obsługiwana przez komponent typu **Timer** i w momencie kliknięcia w klawisz myszki spowodowałaby wykonanie zdarzenia **MojeZdarzenie**, które to zdarzenie „uruchomiło by” odpowiednią metodę zmieniającą treść właściwości **Text** obiektu klasy **label**. Szczegółowo zdarzenia związane z kontrolkami .NET omówione zostaną na następnym wykładzie.



## Podsumowanie

- Delegaty w C# są kolejnym elementem języka zwiększającym jego elastyczność i możliwość pisania efektywnego kodu.
- Delegaty stanowią „źródło” innych elementów języka takich jak : **metody anonimowe**, **wyrażenia lambda**, **język LINQ**, które jeszcze bardziej usprawniają i uelastyczniają pisanie kodu w C#.
- Delegaty stanowią niezbędny element filozofii obsługi zdarzeń w C#.
- Delegaty, same w sobie , nie stanowią jakiegoś istotnego atrybutu obiektowości. Rozwiązania podobne występują też w językach „nieobiektowych”.
- Należy spodziewać się wprowadzenia w nowszych wersjach C# kolejnych rozwiązań usprawniających tworzenia kodu w oparciu o delegaty.

***Dziękuję  
za uwagę 😊***